Why does std::is_copy_constructible report that a vector of move-only objects is copy constructible?

devblogs.microsoft.com/oldnewthing/20190926-00

September 26, 2019

Raymond Chen

The `std::is_copy_constructible` traits class reports whether a type is copy-constructible. But it sometimes reports that a type is copy-constructible even though it isn't.

```
#include <memory>
#include <vector>
#include <type_traits>

// unique_ptr is movable but not copyable.
using move_only = std::unique_ptr<int>;

// This assertion succeeds
static_assert(std::is_copy_constructible_v<std::vector<move_only>>);

// But the type is not copy-constructible.
void f(std::vector<move_only> v)
{
    auto copy = v; // long confusing error message
}
```

The Visual C++ compiler's error message is the most expansive, which doesn't necessarily mean it's the most helpful.

```
xmemory0(819): error C2280: 'std::unique_ptr<int, std::default_delete<_Ty>>::
unique_ptr(const std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': attempting to
reference a deleted function
        with
        [
            _Ty=int
        ]
memory(1968): note: see declaration of 'std::unique_ptr<int,
std::default_delete<_Ty>>::unique_ptr'
        with
        [
            _Ty=int
        ]
memory(1968): note: 'std::unique_ptr<int, std::default_delete<_Ty>>::unique_ptr(const
std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': function was explicitly deleted
        with
        [
            _Ty=int
        ]
xmemory(141): note: see reference to function template instantiation 'void
std::_Default_allocator_traits<_Alloc>::construct<_Ty, _Ty&>(_Alloc &, _Objty *const
, _Ty &)' being compiled
        with
        [
            _Alloc=std::allocator<move_only>,
            _Ty=std::unique_ptr<int, std::default_delete<int>>,
            _Objty=std::unique_ptr<int, std::default_delete<int>>
        ]
xmemory(142): note: see reference to function template instantiation 'void
std::_Default_allocator_traits<_Alloc>::construct<_Ty, _Ty&>(_Alloc &, _Objty *const
, _Ty &)' being compiled
        with
        [
            _Alloc=std::allocator<move_only>,
            _Ty=std::unique_ptr<int,std::default_delete<int>>,
            _Objty=std::unique_ptr<int,std::default_delete<int>>
        ]
xmemory(173): note: see reference to function template instantiation 'void
std::_Uninitialized_backout_al<_Ty *, _Alloc>::_Emplace_back<_Ty&>(_Ty &)' being
compiled
        with
        [
            _Ty=std::unique_ptr<int,std::default_delete<int>>,
            _Alloc=std::allocator<move_only>
        ]
xmemory(173): note: see reference to function template instantiation 'void
std::_Uninitialized_backout_al<_Ty *, _Alloc>::_Emplace_back<_Ty&>(_Ty &)' being
compiled
        with
        [
            _Ty=std::unique_ptr<int,std::default_delete<int>>,
            _Alloc=std::allocator<move_only>
```

```
        ]
vector(1444): note: see reference to function template instantiation '_NoThrowFwdIt
*std::_Uninitialized_copy<_Iter, std::unique_ptr<int, std::default_delete<_Ty>>*,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>>>(const _InIt, const
_InIt, _NoThrowFwdIt, _Alloc &)' being compiled
        with
        [
            _NoThrowFwdIt=std::unique_ptr<int,std::default_delete<int>> *,
            _Iter=std::unique_ptr<int,std::default_delete<int>> *,
            _Ty=int,
            _InIt=std::unique_ptr<int,std::default_delete<int>> *,
            _Alloc=std::allocator<move_only>
        ]
vector(464): note: see reference to function template instantiation
'std::unique_ptr<int, std::default_delete<_Ty>> *std::vector<move_only,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>>::
_Ucopy<std::unique_ptr<_Ty, std::default_delete<_Ty>>*>(_Iter, _Iter,
std::unique_ptr<_Ty, std::default_delete<_Ty>> *)' being compiled
        with
        [
            _Ty=int,
            _Iter=std::unique_ptr<int,std::default_delete<int>> *
        ]
vector(464): note: see reference to function template instantiation
'std::unique_ptr<int, std::default_delete<_Ty>> *std::vector<move_only,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>>::
_Ucopy<std::unique_ptr<_Ty, std::default_delete<_Ty>>*>(_Iter, _Iter,
std::unique_ptr<_Ty, std::default_delete<_Ty>> *)' being compiled
        with
        [
            _Ty=int,
            _Iter=std::unique_ptr<int,std::default_delete<int>> *
        ]
vector(456): note: while compiling class template member function
'std::vector<move_only, std::allocator<_Ty>>::vector(const std::vector<_Ty,
std::allocator<_Ty>> &)'
        with
        [
            _Ty=move_only
        ]
test.cpp(21): note: see reference to function template instantiation
'std::vector<move_only, std::allocator<_Ty>>::vector(const std::vector<_Ty,
std::allocator<_Ty>> &)' being compiled
        with
        [
            _Ty=move_only
        ]
type_traits(694): note: see reference to class template instantiation
'std::vector<move_only, std::allocator<_Ty>>' being compiled
        with
        [
            _Ty=move_only
```

```
        ]
test.cpp(16): note: see reference to variable template 'const bool
is_copy_constructible_v<std::vector<std::unique_ptr<int, std::default_delete<int> >,
std::allocator<std::unique_ptr<int, std::default_delete<int> > > > >' being compiled
```

gcc and clang's errors are roughly comparable. Here's clang:

```
In file included from memory:64:
stl_construct.h:75:38: error: call to deleted constructor of 'std::unique_ptr<int,
std::default_delete<int> >'
    { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
                                     ^   ~~~~~~~~~~~~~~~~~~~~~~~~~~
stl_uninitialized.h:83:8: note: in instantiation of function template specialization
'std::_Construct<std::unique_ptr<int, std::default_delete<int> >, const
std::unique_ptr<int, std::default_delete<int> > &>' requested here
                std::_Construct(std::__addressof(*__cur), *__first);
                     ^
stl_uninitialized.h:134:2: note: in instantiation of function template specialization
'std::__uninitialized_copy<false>::__uninit_copy<__gnu_cxx::__normal_iterator<const
std::unique_ptr<int, std::default_delete<int> > *, std::vector<std::unique_ptr<int,
std::default_delete<int> >, std::allocator<std::unique_ptr<int,
std::default_delete<int> > > > >, std::unique_ptr<int, std::default_delete<int> > *>'
requested here
        __uninit_copy(__first, __last, __result);
        ^
stl_uninitialized.h:289:19: note: in instantiation of function template
specialization 'std::uninitialized_copy<__gnu_cxx::__normal_iterator<const
std::unique_ptr<int, std::default_delete<int> > *, std::vector<std::unique_ptr<int,
std::default_delete<int> >, std::allocator<std::unique_ptr<int,
std::default_delete<int> > > > >, std::unique_ptr<int, std::default_delete<int> > *>'
requested here
    { return std::uninitialized_copy(__first, __last, __result); }
                  ^
stl_vector.h:463:9: note: in instantiation of function template specialization
'std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<const std::unique_ptr<int,
std::default_delete<int> > *, std::vector<std::unique_ptr<int,
std::default_delete<int> >, std::allocator<std::unique_ptr<int,
std::default_delete<int> > > > >, std::unique_ptr<int, std::default_delete<int> > *,
std::unique_ptr<int, std::default_delete<int> > >' requested here
          std::__uninitialized_copy_a(__x.begin(), __x.end(),
               ^
test.cpp:21:13: note: in instantiation of member function
'std::vector<std::unique_ptr<int, std::default_delete<int> >,
std::allocator<std::unique_ptr<int, std::default_delete<int> > > >::vector' requested
here
auto copy = v;
           ^
unique_ptr.h:394:7: note: 'unique_ptr' has been explicitly marked deleted here
      unique_ptr(const unique_ptr&) = delete;
      ^
```

and here's gcc:

```
In file included from memory:65,
                 from test.cpp:2:
stl_construct.h: In instantiation of 'void std::_Construct(_T1*, _Args&& ...) [with
_T1 = std::unique_ptr<int>; _Args = {const std::unique_ptr<int,
std::default_delete<int> >&}]':
stl_uninitialized.h:89:18:   required from 'static _ForwardIterator
std::__uninitialized_copy<_TrivialValueTypes>::__uninit_copy(_InputIterator,
_InputIterator, _ForwardIterator) [with _InputIterator =
__gnu_cxx::__normal_iterator<const std::unique_ptr<int>*,
std::vector<std::unique_ptr<int> > >; _ForwardIterator = std::unique_ptr<int>*; bool
_TrivialValueTypes = false]'
stl_uninitialized.h:142:15:   required from '_ForwardIterator
std::uninitialized_copy(_InputIterator, _InputIterator, _ForwardIterator) [with
_InputIterator = __gnu_cxx::__normal_iterator<const std::unique_ptr<int>*,
std::vector<std::unique_ptr<int> > >; _ForwardIterator = std::unique_ptr<int>*]'
stl_uninitialized.h:305:37:   required from '_ForwardIterator
std::__uninitialized_copy_a(_InputIterator, _InputIterator, _ForwardIterator,
std::allocator<_Tp>&) [with _InputIterator = __gnu_cxx::__normal_iterator<const
std::unique_ptr<int>*, std::vector<std::unique_ptr<int> > >; _ForwardIterator =
std::unique_ptr<int>*; _Tp = std::unique_ptr<int>]'
stl_vector.h:555:31:   required from 'std::vector<_Tp, _Alloc>::vector(const
std::vector<_Tp, _Alloc>&) [with _Tp = std::unique_ptr<int>; _Alloc =
std::allocator<std::unique_ptr<int> >]'
test.cpp:21:13:   required from here
stl_construct.h:75:7: error: use of deleted function 'std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = int; _Dp =
std::default_delete<int>]'
   75 |     { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
      |       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from memory:81,
                 from test.cpp:2:
unique_ptr.h:461:7: note: declared here
  461 |       unique_ptr(const unique_ptr&) = delete;
      |       ^~~~~~~~~~
```

The gcc and clang errors are roughly the same, just in reverse order. gcc's message ordering is frustrating because it spends so much time setting the scene that you want to interrupt it and say *get to the point already I don't have all day*.

Anyway, the deal is that a vector of move-only objects is not copyable because the contents of the vector cannot be copied. So why did `std::is_copy_constructible` say that it was copyable?

Because `std::is_copy_constructible` looks at whether the class has a copy constructor, and `std::vector` has a copy constructor. The copy constructor doesn't compile if you have a vector of move-only objects, but `std::is_copy_constructible` doesn't try to compile the copy constructor. It just checks whether the copy constructor exists.

You can't really expect `std::is_copy_constructible` to try to compile the copy constructor, because the copy constructor's definition may not be visible at the time you ask.

```
struct copyable
{
  copyable();
  copyable(const copyable&);
};
```

The `copyable` class claims to be copyable, but how do we know that its copy constructor will compile successfully? There's no way to know, because there is no definition visible. We have to go by what it says on the tin, and the tin says that it's copyable.

The `std::vector` and other collections claim to be copyable, but whethey they actually *are* copyable depends on what you put into them.

We'll investigate one of the consequences of this "Trust what it says on the tin" behavior next time.

Raymond Chen

Follow